

GPU ENABLED PARTICLE SWARM OPTIMIZATION

A Paper
Submitted to the Graduate Faculty
of the
North Dakota State University
of Agriculture and Applied Science

By
Bala Venkata Rama Kishore Killada

In Partial Fulfillment of the Requirements
for the Degree of
MASTER OF SCIENCE

Major Department:
Computer Science

March 2017

Fargo, North Dakota

North Dakota State University
Graduate School

Title

GPU ENABLED PARTICLE SWARM OPTIMIZATION

By

BALA VENKATA RAMA KISHORE KILLADA

The Supervisory Committee certifies that this *disquisition* complies with North Dakota State University's regulations and meets the accepted standards for the degree of

MASTER OF SCIENCE

SUPERVISORY COMMITTEE:

Dr. Simone Ludwig

Chair

Dr. Saeed Salem

Dr. Ying Huang

Approved:

03/20/2017

Date

Dr. Brian Slator

Department Chair

ABSTRACT

GPUs are massively parallelized devices. They were designed to handle billions of pixels per second. It does this by embracing an incredible level of parallelism. GPUs in fact can leverage any other parallel algorithms developed for super computers. Many disciplines in science and engineering are achieving high speedups on their codes using GPUs. This paper implements a GPU-enabled Particle Swarm Optimization (PSO) algorithm and evaluates the scalability of the algorithm. Several experiments were conducted comparing the CPU with the GPU implementation, analyzing the scalability of the GPU implementation with regards to increases in population size and dimension, as well as memory usage needed. Overall, the results reveal that the GPU implementation of the PSO algorithm scales very well executed on the Nvidia Tesla K40.

ACKNOWLEDGEMENTS

I would like to thank my advisor Dr. Simone Ludwig for giving me this opportunity to work under her guidance, for the immense support and motivation.

I would also like to thank my committee members for taking part in my research and for their support. Huge thanks for my wife Geetha and my family for their constant support and encouragement throughout my graduate studies. Many thanks to the North Dakota State University Graduate School for providing a financial support for my graduate studies.

DEDICATION

Amma.

TABLE OF CONTENTS

ABSTRACT.....	iii
ACKNOWLEDGEMENTS	iv
DEDICATION	v
LIST OF TABLES	viii
LIST OF FIGURES	ix
LIST OF ABBREVIATIONS.....	x
1. INTRODUCTION	1
1.1. Parallel Computing.....	1
1.2. History of Parallel Computing	2
1.3. Particle Swarm Optimization (PSO)	2
1.4. GPU (Multicore processors) and CUDA	3
2. RELATED WORK	4
2.1. CUDA Architecture.....	5
2.2. Thread Batching	7
2.3. CUDA Memory Model	10
3. APPROACH	17
3.1. Implementation.....	17
3.1.1. PSO Algorithm	18
3.1.2. Implementation Details	20
3.2. GPU Implementation.....	21
3.2.1. GPU Code.....	23
3.2.2. Classes	29
3.2.3. Constructors and Destructors	29
3.2.4. Exception Handling in C++.....	31

3.2.5. CUBLAS	32
4. EXPERIMENTS AND RESULTS	35
4.1. Scaling Experiment 1	36
4.1.1. CPU Time Varying p.....	37
4.1.2. GPU Time Varying p.....	38
4.1.3. CPU and GPU Time Comparison Varying p	40
4.2. Scaling Experiment 2	42
4.2.1. CPU Time Varying d.....	42
4.2.2. GPU Time Varying d.....	43
4.2.3. CPU and GPU Time Comparison Varying d	45
5. CONCLUSION AND FUTURE WORK	48
6. REFERENCES	49

LIST OF TABLES

<u>Table</u>	<u>Page</u>
1: CPU time on varying p values	37
2: GPU time on varying p values	39
3: Varying P value CPU vs GPU time with speedup value	40
4: CPU time on varying d values	42
5: GPU time on varying d values	44
6: Varying d value CPU vs GPU time with speedup value	45

LIST OF FIGURES

<u>Figure</u>	<u>Page</u>
1: History of Parallel Computing.....	2
2: Host, Device and Kernel [18]	6
3: Memory Model of CUDA [20].....	9
4: Thread, Block and Grid [18].....	10
5: Caches [22]	11
6: Kepler GK110 Chip Block Diagram - Part of Tesla K40 [21]	12
7: CUDA Device Memory Allocation [22]	13
8: Properties of Memory Types [24].....	16
9: Computational Speed of Different CPUs and GPUs [24].....	22
10: Hardware Design of CPU and GPU [23].....	23
11: Ackley Function Graphical Representation [26]	36
12: CPU Time Versus Number of Particles with Varying P	38
13: GPU Time Versus Number of Particles on Varying P	39
14: CPU vs GPU Time Performance with Varying P Value	41
15: Speedup of GPU Versus CPU with Varying P Value	41
16: CPU Time Versus Number of Dimensions on Varying D.....	43
17: GPU Time Versus Number of Dimensions on Varying D Values	44
18: CPU vs GPU Time Performance with Varying D Value.....	46
19: Speed Up of GPU Versus CPU on Varying D Value	47

LIST OF ABBREVIATIONS

GPU.....	Graphics processing unit.
PSO	Particle swarm optimization.
CPU.....	Central processing unit.
MPP.....	Massively Parallel Processing.
CUDA	Compute Unified Device Architecture.
GPGPU	General-purpose graphics processing unit.
API.....	Application program interface.
Nvcc	Nvidia CUDA Compiler.
OS	Operating system.
RAM	Random access memory.
GB	Giga bytes.
TPB	Threads-per-block.
CUBLAS.....	CUDA Basic Linear Algebra Subprograms.

1. INTRODUCTION

Most of the Universe is built-in parallel, with various things going on simultaneously. Every person or organization want to do work they are capable of but how effective the work can be depending on the process they choose. Suppose we have a lot of work to be done, and we want the work to be done even faster, then we would distribute it among multiple persons or organizations.

1.1. Parallel Computing

Given work is divided into multiple workloads using the divide and conquer technique and processing each one of them on different CPUs is the basic idea of parallel computing. The method of parallel computing can be explained in many ways. If the work is having separate jobs that are not dependent on each other and they all take the same time, then we will have the fastest result. But if the jobs take widely different amount of time and no relation with each other then we need to assign to every system equally. If one system completes first, then it needs to help another and this whole process works fairly well [1].

For example, we use “Multiple Queue Multiple Server” in grocery shops. “Single Queue Multiple Server” are systems used in banks, pizza parlors, printing shops, etc. The above cases explain the main scenario of the parallel computing in different situations. Finally let us assume that the work we have is only a single job but takes a very long time to complete the reorganizing of the job and breaking it into different jobs that can be easily done in parallel. The parallel programmer should be able to break the problem into parts and has to figure out how the parts relate to each other [1].

1.2. History of Parallel Computing

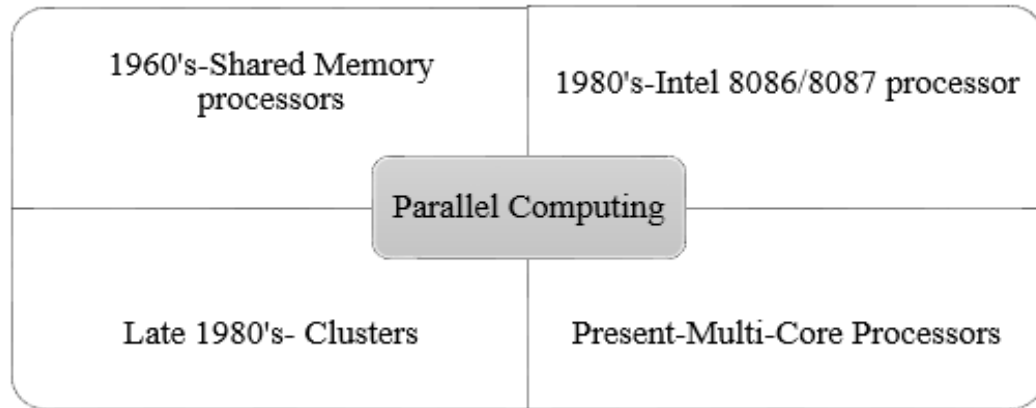


Figure 1: History of Parallel Computing

Parallel computing interest began in the 1960's and 1970's where the machines started using the shared memory multiprocessors using the shared data with multiprocessors working side by side. Later, during the mid-1980's came the concept of Massively Parallel Processors (MPPs) where extreme performance could be achieved with huge data. Figure 1 illustrates the history of parallel computing.

Clusters are computers operated in parallel using a large number of computers connected by a network. Clusters help us to gain substantial cost savings as the more cores we add the more we save. We use clusters to perform more jobs in less time. Low cost per job is also achieved with the use of clusters [2]. Later on, the introduction to advanced GPU's in the computing field has led to tremendous results as expected.

1.3. Particle Swarm Optimization (PSO)

Particle swarm optimization is proposed by James Kennedy and Russell Eberhart in 1995. It is inspired by social behavior of birds and fishes. It combines self-experience with social experience and is known as a "Population based optimization". It uses a number of particles that constitute a swarm moving around in the search space looking for the best solution. Each particle

in the search space adjusts its flying according to its own flying experience as well as with the flying experience of other particles. In a swarm, each individual represents a solution and each solution corresponds to a fitness value. The whole swarm is likely to converge at an optimal solution.

1.4. GPU (Multicore processors) and CUDA

Graphical Processing Units (GPUs) are completely framed to enhance the graphics and compute frame to be displayed on the screen. The idea of using this powerful resource for generic computations where parallel processing is used for large amount of data has given satisfying results. GPUs constitute hardware that are very good at certain type of mathematical operations. The architecture has lots of identical computing units determining similar mathematical functions such as waving blades of grass displayed on the screen. GPGPU is introduced for using the GPU for general purpose computing with different programming languages, which involves huge data compilations. Being numerous, the same calculations are often applied to many data items one after the other. So, the GPU is designed to perform a very large number of same or similar calculations on data items very quickly. Programming for GPU is nothing like programming for a CPU, even though with CUDA C (Compute Unified Device Architecture) the code structure looks similar. A CPU is a small collection of very powerful cores whereas a GPU is a large collection of only modestly powerful cores [3].

Nvidia developed the CUDA technology to implement GPU based applications more easily. It is a hardware and software architecture, which runs under the C language environment for managing computations on GPU as a data parallel-computing device. The programming model and instruction set architecture helps to solve complex solutions more efficiently than a CPU.

2. RELATED WORK

The main idea of multithreading GPU is to show the possibility to significantly improve the PSO algorithm performance with simple and almost straightforward parallel programming. Other parallelization techniques applied to the particle swarm optimization algorithm include using Hadoop MapReduce where all nodes in the cluster work on the same population and each node in cluster has its own portion of the population. Another parallelization technique is MPI (Message Passing Interface). MPI relies on thread-based programming of the code to be parallelized.

GPU enabled related work is reviewed as our approach is related to other GPU implementations. These research papers make use of CUDA [4]. An approach to GPU-based parallel multi-objective PSO algorithm was proposed by [5] in which experiments on four two-objective benchmark functions were conducted and achieved results of speedups up to 10. Similarly, Adaptive Mutation PSO algorithm is proposed in [6]. The GPU-based approach helps to coordinate the global and local search ability by adjusting the learning parameters during the optimization run. Speedups of up to 11 are achieved using four different benchmark functions for the experiments. In [7], a GPU-based asynchronous PSO algorithm was proposed conducting the evaluation on five benchmark functions without introducing explicit synchronization mechanism among the particle evolution process.

Another GPU-PSO implementation was designed to solve constraint satisfaction problems [8]. Speedups up to 4 are achieved by the experimental results. Later the evaluation of 3D Pose estimation achieved a speedup of 140 for the application of a GPU-enabled PSO applied to Pose estimation [9]. A model to inverse rendering of 3D models [10] was applied by a similar GPU based PSO algorithm. For a 1D heat conduction equation in [11] a parallel hybrid PSO was

applied and speedups of up to 25 were achieved by the experiments. The use of Canonical PSO algorithm and parallelizing them using GPU CUDA code was closely related to the work and is outlined in [12], [13], [14], [15]. Each of the implementations are slightly different in the way the different stages of PSO algorithm are merged into kernel functions as well as the data structures used in global memory.

2.1. CUDA Architecture

CUDA is a latest programming model, which is used for instructing newer Nvidia GPUs. Graphic cards serve the purpose of fast 3D calculations. These calculations are often simple, but rendering a 3D environment takes millions of calculations every frame. Hence, GPU performs a large number of similar calculations on data items very quickly. CPU is a small collection of powerful cores, whereas GPU is a large collection of modestly powerful cores. The CUDA compiler will compile all the CUDA code to Parallel Thread Execution (PTX) and sends the remaining C++ code back to the C++ compiler to compile.

The structure of a basic CUDA programming consists of many functionalities. The two key components of GPUs are “threads” and “memory”. They enable us to improve the performance of a CUDA program. Performance is the main component to consider in CUDA programming.

Host, device and kernel are the primary components to consider in a CUDA program. Host code runs on the CPU whereas the CUDA kernels are GPU functions those run on GPU devices. An application starts by executing the code on a CPU host. At a certain point in the program, the host code invokes a GPU kernel on a GPU device [16].

CPU is referred to as ‘Host’. System RAM and the peripheral devices are all in control of the host. It has phases, which exhibit little or no data parallelism. The host performs the following operations:

- initializes the device
- allocates and initializes the input arrays in host DRAM
- allocates memory on the device
- uploads input data to the device
- executes kernel on the device
- downloads the results
- checks the results
- clean-up

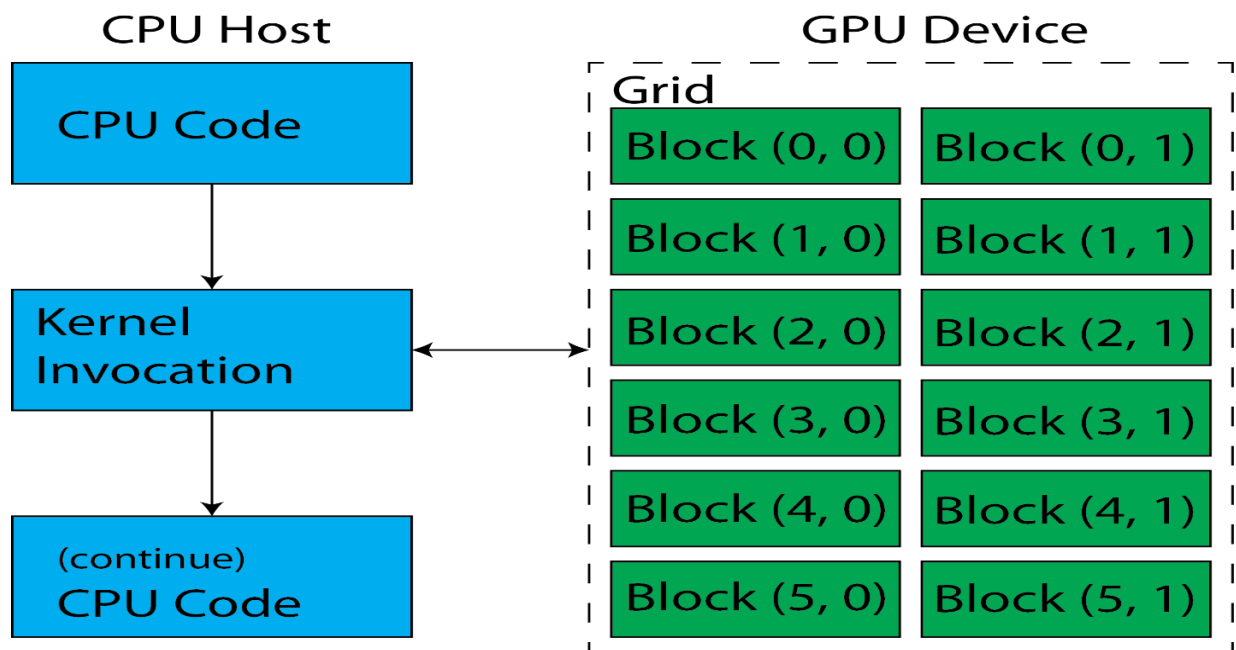


Figure 2: Host, Device and Kernel [18]

The GPU is referred to as 'device'. It has its own RAM and cannot access the system's RAM, disks nor other peripheral devices. It has the phases, which exhibit rich amount of data parallelism. The nvcc compiler further compiles the GPU code. Figure 2 shows the outline of CPU and GPU with the kernel.

A kernel is another major component in a CUDA program. A GPU has 100's or 1000's of threads running simultaneously, which are written in C. The CPU uses a special syntax to launch the kernel, and gives information to the GPU to let it know how many threads should be used. Kernels cannot use system memory [17].

GPU grid is composed of independent groups of threads called thread blocks. The kernel is executed on this GPU grid. The GPU executes a kernel in parallel using many threads. Once the kernel finishes its execution part, the CPU will continue to execute the original program. To execute the files with .cu extension, we place both kernel and host in the same file. To compile and link multiple files, we can use the nvcc (Nvidia CUDA compiler). CUDA C extensions allow the programmer to define kernels, that when called, are executed 'n' times in parallel by 'n' different threads on the GPU devices [17]. A kernel code is only executable on the device. The `__global__` qualifier identifies it with void return type. The CPU executes the host codes and can call GPU kernels. The process of calling a kernel is typically referred to as kernel invocation. The host code uses the same syntax as any other C/C++ program.

2.2. Thread Batching

In order to take advantage of the multiple multiprocessors, kernels are executed as a grid of threaded blocks. All threads in a thread block are executed by a single multiprocessor. The resources of a multiprocessor are divided among the threads in a block (registers, shared memory, etc.). The assignment of the number of threads per block, the number of thread blocks

per grid should be done carefully, so that we can make the maximum utilization of the available resources. Load latency in device memory reads is caused if there are less number of threads per block. Also, just one block per multiprocessor will make the multiprocessor idle during thread synchronization. Therefore, we should have at least twice as many blocks as there are multiprocessors in the device (the number of blocks per grid should be at least 100). Also, assign the number of threads per block in multiples of the warp size, because it lessens the under-populated wraps.

- Threads: single execution units that run kernels on GPU.
- Block: is a collection of threads; all the threads in any single thread block can communicate.
- Grids are collection of blocks.

The kernel is launched as a collection of thread blocks. Each GPU has a limit of the number of threads in a block, but there is almost no limit on the number of blocks. Therefore, each GPU can run some number of blocks concurrently, executing some number of threads simultaneously. Thus, they reduce the workload quicker with absolutely no change to the code. The threads, blocks and grids are executed in different ways respectively. Figure 3 shows a single thread, block and grid and the memory they use.

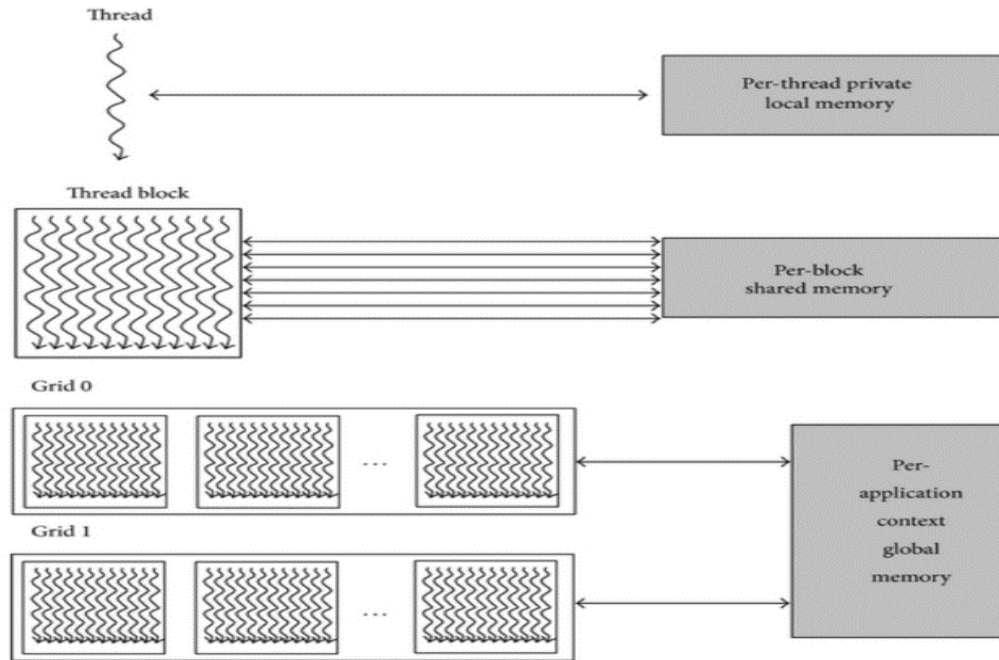


Figure 3: Memory Model of CUDA [20]

With respect to the Nvidia's GPU chip, the following terms are used:

- **Thread:** Each MP is further divided into several stream processors (SPs), with each SP handling one or more threads in a block.
- **Block:** The GPU chip is technically a collection of multiprocessors. Every single multiprocessor is responsible for handling one or more blocks in a grid. A block is never divided across multiple multiprocessors, only one multiprocessor takes care of a block.
- **Grid:** An entire grid is handled by a single GPU chip.

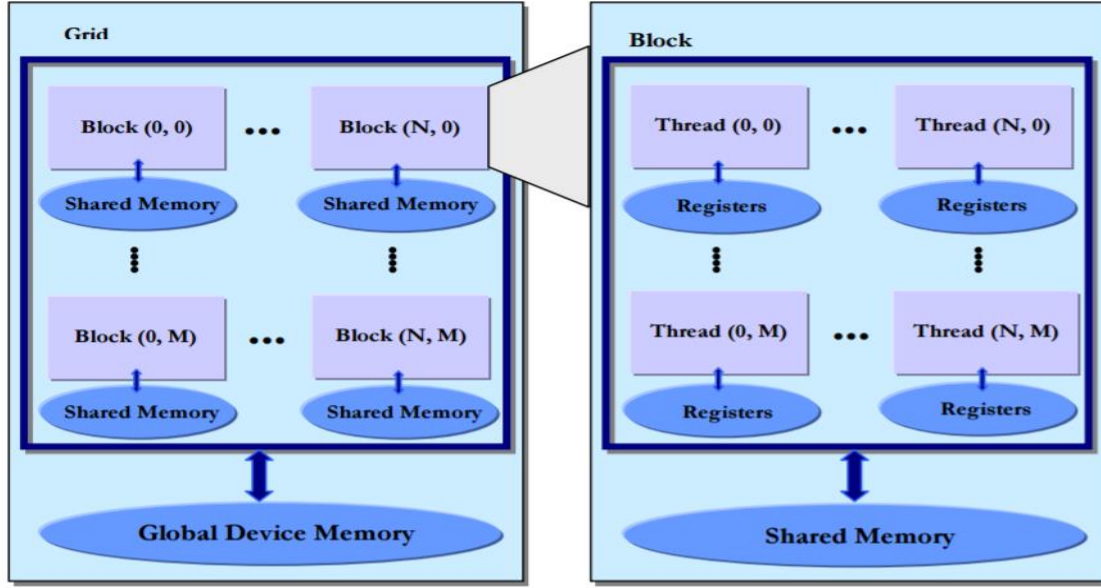


Figure 4: Thread, Block and Grid [18]

Figure 4 shows the inside view of a grid which is like a collection of blocks and the view of a block which is collection of threads. At runtime, a thread can determine the block that it belongs to, the block dimensions, and the thread index within the block. These values can be used to compute indices into input and output arrays. We must have at least as many thread blocks as there are multiprocessors, otherwise some multiprocessors will be dormant during the kernel execution. The number of blocks should be a multiple of the number of multiprocessors for balanced work distribution. In practice, there should be hundreds or thousands of threads, blocks to ensure full occupancy on current and future hardware. Each thread block should contain a multiple of 32 threads (32 being the warp size), otherwise the multiprocessors are underutilized – 256 is a good number, 512 is maximum.

2.3. CUDA Memory Model

A CUDA device has many different memory components, each with a different size and bandwidth. To effectively read and write to a specific memory component, we need to know how the memory is organized on these components. Ideally, a program would be structured so that

threads do not need to constantly go to global memory to retrieve data, which is ineffective and slow.

For compute capability 2.0 and up there is an L1 cache per multiprocessor. There is also an L2 cache which is shared between all multiprocessors. The global local memory uses these. L1 is a very fast, shared memory store. L1 and shared memory are the same bytes, they can be configured to be 48k or shared as 16k of L1 or 16k of shared and 48k of L1. All global memory accesses go through the L2 cache, including those from the CPU. The texture and constant memory have their own separate caches.

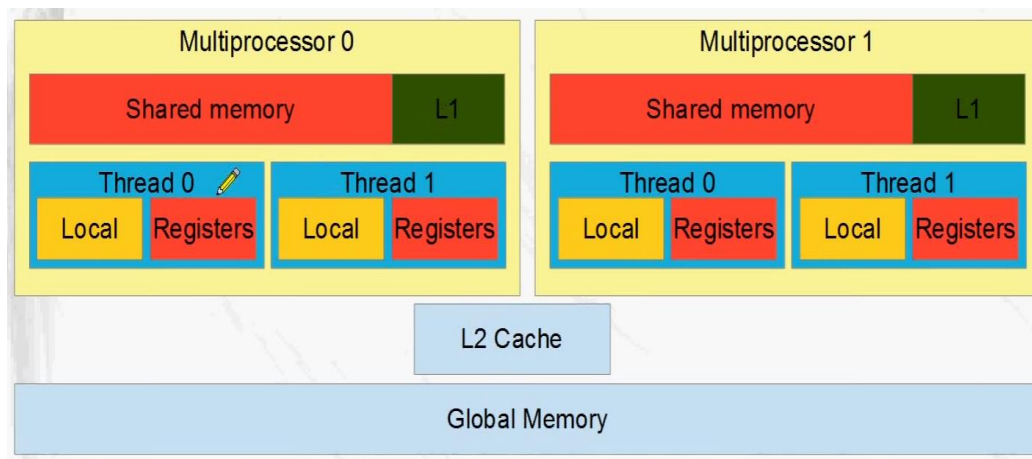


Figure 5: Caches [22]

As we see in Figure 5, the local memory is often not inside the block, it is often in the same area as global memory. In the CPU architecture, the L1 cache is faster than the L2 cache, and L3 is the slowest cache, but still much faster than main memory. Similarly, a GPU has several levels of cache memory to reduce read/write latency. Accessing memory from global memory relies on random access, which might take hundreds of clock cycles. Accessing memory from cache is paged-locked, which only takes a few clock cycles. The size of the different memory components also needs to be considered. Global memory is significantly larger than

cache memory and registers are much smaller. The Kepler GK110 Chip block diagram that is part of Tesla K40 [21] (that we use for the experiments) is shown in Figure 6.



Figure 6: Kepler GK110 Chip Block Diagram - Part of Tesla K40 [21]

Memory Access:

We use one and two-way arrows to indicate the read (R) and write (W) capability. An arrow pointing toward a memory component indicates write capability; an arrow pointing away from a memory component indicates read capability. For example, global memory and constant memory can be read (R) or write (W). On a CUDA device, multiple kernels can be invoked. Each kernel is an independent grid consisting of one or more blocks. Each block has its own per-block shared memory, which is shared among the threads within that block. All threads can access (R/W) different parts of memory on the device that are summarized below.

In general, we use the host to transfer data to and from global memory and to transfer data to and from constant memory. Once the data is in the device memory, our threads can read and write (R/W) to different parts of memory:

- R/W per-thread register
- R/W per-thread local memory
- R/W per-block shared memory
- R/W per-grid global memory
- R per-grid constant memory

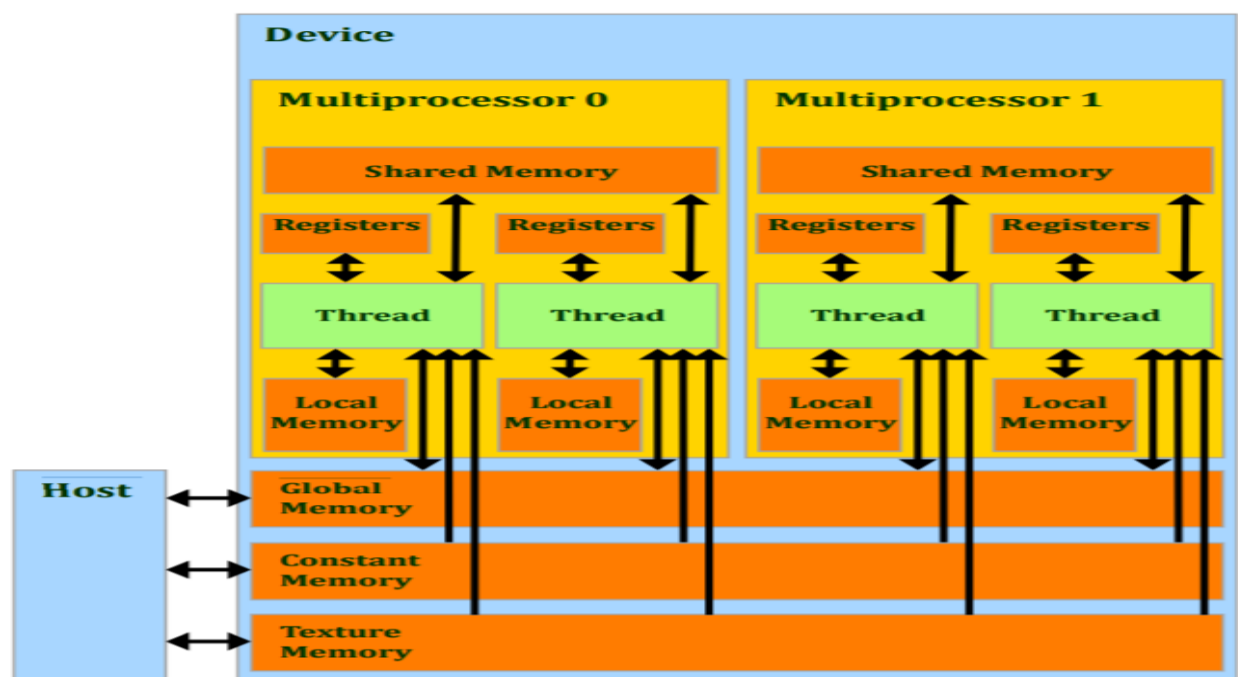


Figure 7: CUDA Device Memory Allocation [22]

On devices with compute capability 1.x, there are 2 locations where memory can possibly reside; cache memory and device memory. The cache memory is considered “on-chip” and accesses to the cache is very fast.

Shared memory and cached constant memory are stored in cache memory with devices that support compute capability 1.x. The device memory is considered “off-chip” and accesses to

device memory is about $\sim 100\times$ slower than accessing cached memory. Global memory, local memory and (uncached) constant memory are stored in device memory. On devices that support compute capability 2.x, there is an additional memory bank that is stored with each streaming multiprocessor. This is considered L1-cache and although the address space is relatively small, its access latency is very low.

A CUDA device has several different memory components that are available to programmers - register, shared memory, local memory, global memory and constant memory. Figure 7 illustrates how threads in the CUDA device can access the different memory components. In CUDA, only threads and the host can access memory. There are several different types of memory that your CUDA application has access to. For each different memory type there are tradeoffs that must be considered when designing the algorithm for your CUDA kernel. Global memory has a very large address space, but the latency to access this memory type is very high. Shared memory has a very low access latency but the memory address is small compared to Global memory. Figure 8 shows the comparison between different types of memories.

- Register memory access is very fast, but the number of registers that are available per block is limited. It is currently not possible to perform random access to register variables. Register variables are private to the thread. Threads in the same block will get private versions of each register variable. Register variables only exist as long as the thread exists. Once the thread finishes execution, a register variable cannot be accessed again. Each invocation of the kernel function must initialize the variable each time it is invoked. Variables declared in register memory can be both read and written inside the kernel. Reads and writes to register memory do not need to be synchronized [23].

- Each thread has private (or local) memory it is used for local variables of the thread. Private memory is first allocated in registers (there are 16K registers in a thread block, they are used for all the threads). If the threads need more private memory than there are registers, local memory is spilled to global memory with serious performance consequences.
- Threads in a thread block share a shared memory (programmable cache). The program explicitly declares variables (usually arrays) to live in shared memory. Because it is on-chip, shared memory is much faster than local and global memory, but slower than to registers. Shared memory latency is roughly 100x lower than global memory latency.
- Team work in thread block. Different threads may read different elements into shared memory, but all threads can access all shared memory locations. Hence, it is extremely fast, highly parallel and restricted to a block (private to each block). We use this in e.g. matrix multiply.
- Global memory is the best memory if 68 or 128 bytes (16 or 32 words) are read. It allows parallel read/writes from threads in a block, sequential memory locations. It allows helps us with proper alignment, called “coalesced” read/write. Otherwise, it needs a sequence of reads/writes, which is more than 10 times slower. All threads in all thread blocks can access all global memory locations. Global memory is persistent across thread block activations, kernel calls. Global memory has a lifetime of the application and is accessible to all threads of all kernels. One must take care when reading from and writing to global memory because thread execution cannot be synchronized across different blocks. The only way to ensure access to global memory is synchronized is by invoking

separate kernel invocations (splitting the problem into different kernels and synchronizing on the host between kernel invocations) [24].

Memory	Located	Cached	Access	Scope	Lifetime
Register	cache	n/a	Host: None Kernel: R/W	thread	thread
Local	device	1.x: No 2.x: Yes	Host: None Kernel: R/W	thread	thread
Shared	cache	n/a	Host: None Kernel: R/W	block	block
Global	device	1.x: No 2.x: Yes	Host: R/W Kernel: R/W	application	application
Constant	device	Yes	Host: R/W Kernel: R	application	application

Figure 8: Properties of Memory Types [24]

- Like global variables, constant variables must be declared in global scope (outside the scope of any kernel function). Constant variables share the same memory banks as global memory (device memory) but unlike global memory, there is only a limited amount of constant memory that can be declared (64KB on all compute capabilities). Access latency to constant memory is considerably faster than global memory because constant memory is cached but unlike global memory, constant memory cannot be written to from within the kernel. This allows constant memory caching to work because we are guaranteed that the values in constant memory will not be changed and therefore will not become invalidated during the execution of a kernel [24].

3. APPROACH

Particle Swarm Optimization (PSO) involves several particles exploring a search space for optimal parameters. The optimal parameters are identified using an objective function that associates a cost to each of the parameters. A GPU implementation provides a parallel processing solution to utilize many core in a processor to accomplish the optimization task. Implemented is the PSO algorithm utilizing NVidia's CUDA.

The PSO algorithm is implemented as follows. Particles are randomly distributed throughout a map in memory. In this case, this distribution sets the seed to the clock time in seconds into a 64-bit initial seed value, which is added to 1011, the particle location index, and -1. This is then fed into a function called `_pso_rnd`, which multiplies the seed by a large prime value [402653189], which then adds [805306457] another large prime number to the 64bit seed. The random value is returned via a pointer, which the function returns the seed divided by a large number, the largest float plus one.

3.1. Implementation

In the parallel implementation, the communication between processors is the main performance holdup. The minimum communication between different computational nodes ensures that iterations are not dependent and many resources are shared between the processors or nodes. The client machine takes care of all the communication in Matlab. The algorithm describes that the iterations to be given to the nodes and defined before the execution. The fitness functions take the role for optimizing the position, local best, global best values, which are the resources between the threads, and are chosen by Matlab. Including the three properties, which are the number of particles, dimensions, and the number of iterations, the number of nodes is to be considered by these three factors.

3.1.1. PSO Algorithm

A numerical vector of D dimensions, usually randomly initialized in a search space, is conceptualized as a point in a high-dimensional Cartesian coordinate system. Because it moves around the space testing new parameter values, the point is well described as a particle. Because a number of them (usually $10 < N < 100$) perform this behavior simultaneously, and because they tend to cluster together in optimal regions of the search space, they are referred to as a particle swarm.

The following is a introduction to the operation of the particle swarm algorithm. Consider a flock or swarm of p particles, with each particle's position representing a possible solution point in the design problem space D. For each particle i, Kennedy and Eberhart proposed that the position x^i be updated in the following manner:

$$x_{k+1}^i = x_k^i + v_{k+1}^i \quad (1)$$

With a pseudo-velocity v_{k+1}^i calculated as follows:

$$v_{k+1}^i = v_k^i + c_1 r_1 (p_k^i - x_k^i) + c_2 r_2 (p_k^g - x_k^i) \quad (2)$$

Here, subscript k indicates a (unit) pseudo-time increment, p_k^i represents the best position of particle i at time k (the cognitive contribution to the search vector), and p_k^g represents the global best position in the swarm at time k (social contribution), r_1 and r_2 represent uniform random numbers between 0 and 1. To allow the product $c_1 r_1$ or $c_2 r_2$ to have a mean of 1, Kennedy and Eberhart proposed that the cognitive and social scaling parameters c_1 and c_2 be selected such that $c_1 = c_2 = 2$. The result of using these proposed values is that the particles overshoot the target half the time, thereby maintaining separation within the group and allowing for a greater area to be searched.

The serial PSO algorithm as it would typically be implemented on a single CPU computer is described below, where p is the total number of particles in the swarm. The best ever fitness value of a particle at design coordinates p_k^i is denoted by f_{best}^i and the best ever fitness value of the overall swarm at coordinates p_k^g by f_{best}^g . At the initialization time step $k=0$, the particle velocities v_o^i are initialized to random values within the limits $0 \leq v_o \leq v_o^{max}$. The vector is calculated as a fraction of the distance between the upper and lower bound.

The algorithm shown below is explained this way, considering a swarm of particles in the space, each particle is initialized with a position value for the purpose of our experiment (these are considered as local best values), given an initial velocity at that particular position for each particle. At the initial positions (local best values), the position of a particle close to the goal is considered as global best value. Usually these positions are initialized to uniformly cover the search space.

1. Initialize the values for p (no. of particles), d (Dimension), IL (no. of iterations)

2. Optimize

- 2.1. Benchmark function

- (a) Select one of the 7 Benchmark functions

- 2.2. Parallelize

- (a) Randomly initialize particle positions $X_0^i \in D$ in IR^n for $i = 1, \dots, p$

- (b) Randomly initialize particle velocities $0 \leq v_0^i \leq v_0^m$ for $i = 1, \dots, p$

- (c) Evaluate function value using design space coordinates

- (d) If $f_k^i \leq f_{best}^i$ then $f_i^{best} = f_k^i, p^i = x_k^i$
- (e) If $f_k^i \leq f_{best}^g$ then $f_{best}^g = f_k^i, p^g = x_k^i$
- (f) If stopping condition is satisfied then go to Step 2.3
- (g) Update particle velocity vector v_{k+1}^i using Eq. (2)
- (h) Update particle position vector x_{k+1}^i using Eq. (1)
- (i) Increment I; if $i > p$ then increment k, and set $i=1$
- (j) If stopping condition is satisfied for the fitness function then go to Step 3
- (k) Go to Step 2.1(a)

2.3 Update Results

3. Update Global best

4. Terminate

3.1.2. Implementation Details

The PSO code is made to run on a Tesla K40 GPU Linux OS machine where CUDA is used for GPU implementation, and general Matlab functions are used for the normal CPU implementation. The inputs we used for this implementation are the number of iterations, number of dimensions, and number of particles.

Test.m

The main function Test.m calls all other functions from within. Inputs are initialized at the beginning; which is needed for the PSO code. We need to specify the number of workers, and to run in parallel. Next, the needed workers are called by the client program and lastly the workers should be released. The data to be shared between workers can be optimized using Matlab that works in a simplified model for parallel processing.

PSO fun.m

The velocities and positions of the particles are initialized in this file and achieve the global best position by comparing all the particles in a run, and then the process continues in a loop.

Test func.m

For the population based search techniques, researchers often depend on empirical studies to scrutinize the behavior of an algorithm. Benchmark problems and real life problems are used for this. For the present study, we have taken the Ackley benchmark function to analyze the PSO algorithm behavior.

3.2. GPU Implementation

Figure 9 shows the computational speed of different CPUs and GPUs. CPU and GPU can take different type of workloads. CPUs can handle different type of fast serial processing and multitasking and the GPUs are useful for high computational throughput with their parallel architectures. GPUs were originally designed to render graphics, they work very well for shading, texturing, and rendering the thousands of independent polygons that comprise a 3-D object. CPUs, on the other hand, provide much more control over the logical flow of a program. GPUs feature more processing cores and higher aggregate memory bandwidth and CPUs have more sophisticated instruction processing and higher clock speed. In Figure 9, almost any

application that relies on massive floating-point operations can gain significant speedup using GPUs. This is typically referred to as a General-Purpose computing on Graphic Processing Unit (GPGPU). For example, routines such as matrix manipulation, fast fourier transforms, and vector operations, can be accelerated using GPUs. Figure 9 shows the differences in hardware design between CPUs and GPUs. It does not accurately represent the actual hardware designs of CPUs and GPUs. There are fewer CPU cores, but they are more complex than the GPU cores. CPUs have larger cache than what is available with the GPUs.

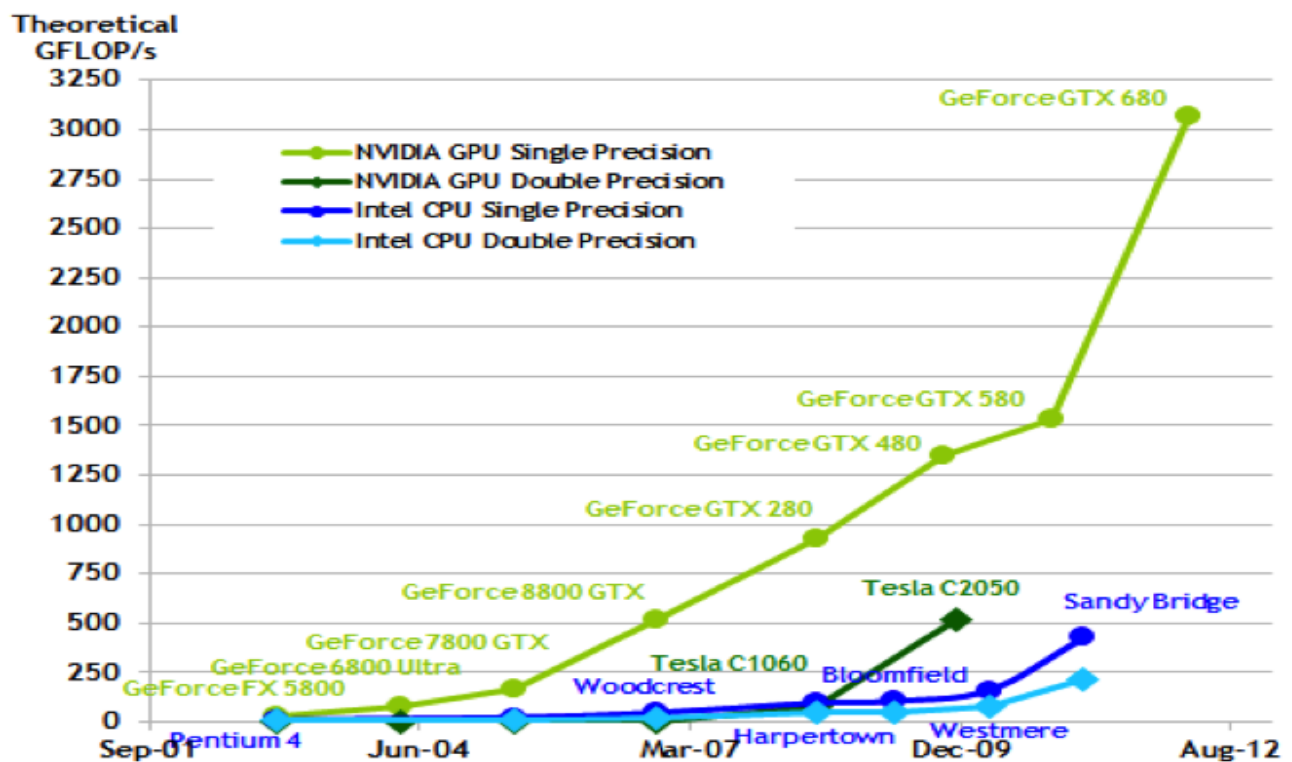


Figure 9: Computational Speed of Different CPUs and GPUs [24]

GPU applications typically adopt the heterogeneous parallel programming model to fully utilize their parallel computing capability. In a heterogeneous computing system, massive

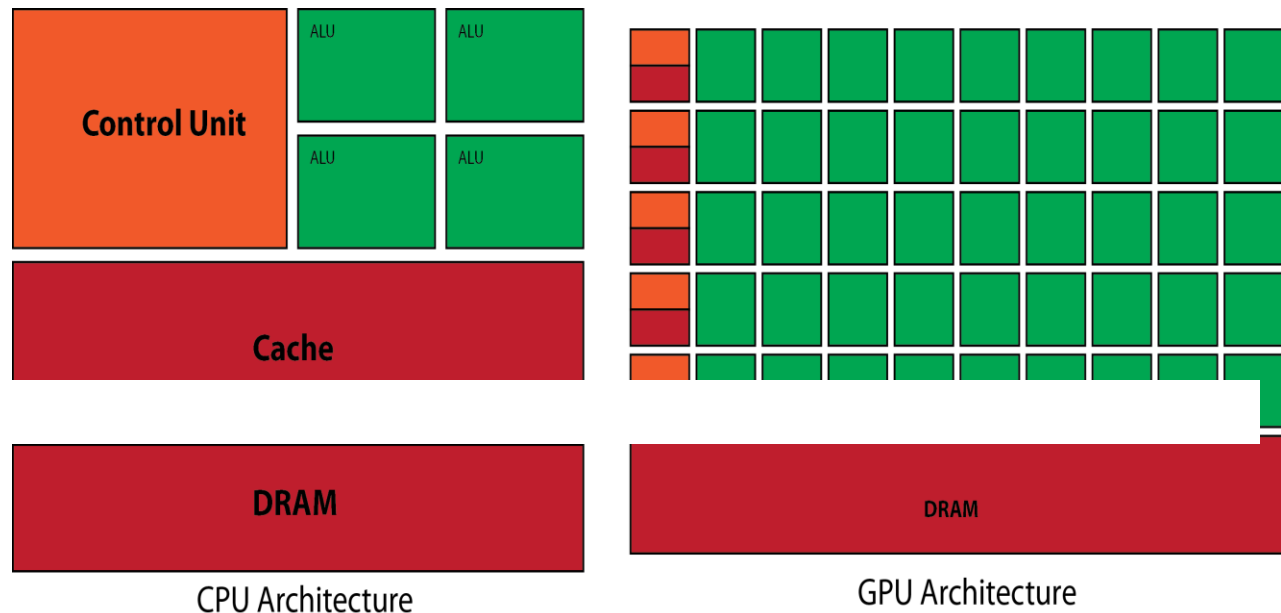


Figure 10: Hardware Design of CPU and GPU [23]

parallel tasks and floating-point intensive tasks are typically offloaded to a GPU-based accelerator like Nvidia Tesla, or a CPU-based parallel coprocessor like the Intel Phi Coprocessor. This allows the code to take advantage of the highly parallel hardware design to produce higher computational throughput. Figure 10 shows the hardware design of the CPU and GPU.

3.2.1. GPU Code

Most GPU tasks are solved in 3 steps:

- Copy input data into GPU memory.
- Run a function on GPU.
- Copy result data from GPU memory back to usual RAM.

To explain the functioning of a GPU, first let us understand to create two strings (internally - 2 arrays with elements, placed flat in memory). To make the code look simpler than pure CUDA C, it is advised to write helper functions which are templates and can be used with different data types like double int, long int, cudaMalloc gives GPU memory and pointer to it. In principle, we can use cudaMemcpy as is; GPU memory pointers are like usual RAM pointers, you can add, subtract something from them. Helpers simply allow doing several steps in one call:

- check requested size
- allocate
- report error
- return ready pointer

For copying data from the RAM to GPU in which cu_memcpy_to_dev / to_host copy contiguous data arrays from RAM to GPU memory and back. The syntax below call copies n floats from string a into GPU memory (destination starting at address pa).

Syntax1: *cu_memcpy_to_dev (pa, &a [0], n)*

- pa - destination on GPU
- &a [0] - address of the first element in a - same as any usual array pointer.

Syntax2: *kern_a_pluseq_b<<<1, int(n)>>>(pa, pb, int(n));*

This syntax <<< >>> is Nvidia extension to C language. It means:

- load GPU machine code, corresponding to kern_a_pluseq_b function, into GPU memory.
- create a short memory block on GPU, and copy arguments pa, pb, int (n) to GPU.
- run 1 block, containing n threads (<<<1, int(n)>>>).

It is necessary to explicitly convert data types to avoid compiler warnings and errors. For example, `n` variable is type `size_t`, which is probably alias to unsigned long long int on your machine. Nvidia <<< >>> extension awaits int or unsigned int (it may be 32-bit value, while long long is 64-bit). We do not currently use arrays >2 GB size, so a 32-bit int is enough.

GPU runs threads in "blocks". Each block usually can contain up to 1024 threads, but in practice, up to 512 are most often used. The block of threads is an "atomic" unit for GPU, i.e. GPU executes threads in block partially in parallel, partially sequentially, until the block is completed, or another block is scheduled. So, when you need to process arrays containing some number of elements, which is not known and may be larger than, for example, 1024, you will use more than one block. Nvidia GPUs are not executing kernel calls (<<<>>> functions) at once where you have written them. GPU driver puts the code into internal queue. The code is executed when one of the following occurs:

- Try to read results (call to `cudaMemcpy` from device to host)
- Explicitly call `CUDA Device Synchronize ()`.

The second case is useful if you want to measure the time for each GPU kernel execution from within the program. The CUDA toolkit contains a profiler, and it is very useful, but it is often necessary to measure time from within C or C++ itself. For each thread to have a unique number, in addition, block and grid dimensions are considered. These are defined once and used with all GPU kernels to write. GPU performance does not depend on grid or block dimensions, only on threads-per-block (TPB). The larger GPU function (the more variables), the smaller will be the tpb, because memory for variables is shared between all threads in a block.

Syntax 3: *bool _realloc (size_t n2) void assign*

Now consider `_realloc`. If new array size is different it allocates an area of such size, and if successful, it frees the previous area. With the help of `_realloc`, the following 3 functions are doing several useful things:

- `assign ()` copies data from a string (i.e. from host RAM).
- `extract ()` copies data back from device to a string.
- `operator=` allows to write smooth like structures such as "`x1 = x2;`", where `x1` and `x2` are both `gpu_array`. The result will be copying data from `x2` to `x1` without redefining `operator=`, copying would duplicate `n` and `pdata` members. And the previous `x1.pdata` would be erased and not freed by `x1` destructor, but `x2.pdata` (duplicated) would be freed by both `x1` and `x2` destructors. This is an error and should be avoided in all similar cases.

The `operator=` functions for allocating new storage: if it fails, it generates an exception, and the program will exit (if the exception is not caught). Next, if the device memory has been allocated successfully it copies data from the source array. Again, if copying fails for any reason (on the GPU side) an exception is generated. If source does not contain data (0 elements), no copying at all occurs. `gpu_array`, surely, must contain more functions, if you want to reuse it in many programs. For example, a function for clearing the GPU array (like `_realloc (0)`), a function for setting a value to all elements in the array, etc. In general, to make `gpu_array` behave exactly like a standard string or vector, caching the GPU data in RAM on demand and accessing by subscript operator. C and C++ are more flexible than most other languages because they allow to work directly with RAM (pointers). But another side is that you always should "manually" control how the memory is allocated and freed. In this relation, C is not convenient because you should call `free ()` (or `cudaFree ()` in case of GPU) explicitly. C++ allows hiding all

this for the account of destructors. So, in C++ you are always responsible for the way the data is copied or passed into a function - "by reference" or "by value". In languages like Java, C#, python, "objects" are always internally pointers and passed most frequently by reference to avoid performance loss in most programs. The language interpreter in such languages hides its destructor-like behavior. This results in the following: a programmer does not care but does not know when the object is destroyed and the memory is freed. When you write, programs demanding high performance and timely memory freeing, C/C++ is preferable in more than fifty percent of the cases.

GPU functions are usually taking pointers and other low-level arguments. And Nvidia prefers making less functions, while making each of them more "functional". Aliases are to make the PSO algorithm code shorter and clearer. `kern_multimin`, `kern_multiavg` uses single function (in parallel) to find the minimum and average values for multiple arrays, residing in one flat array in GPU memory. Optimal processing and efficient calculation of minimums, maximums, averages by multiple threads is called "reduction".

In parallel computation, a single scalar division may take not less or even more time than calculating several sums for pairs of elements. Because, when one thread divides others, it will wait until this is complete. Factually reduction can be used only with less than or equal to 32 threads at once due to the hardware architecture. To call functions such as `sum`, `minimum`, `maximum`, etc., for an arbitrary number of elements we must "manually" cascade less than or equal to 32-sized blocks. This is done with `g_multimin`, `g_multiavg`. Each kernel run can calculate a minimum for maximum of 32 elements. The elements are accessed using the `stride` parameter, which grows exactly by 32 at each iteration because we have to calculate `min()` for several columns (or rows), this procedure calls GPU kernels in such way that each `min()` is

calculated separately. But inside, the reduction is used. The overall time at best might be $\log_2(M)$, where M is size of the row or column count in which $\min()$ is calculated. Time is usually larger and for certain matrix size it is proportional to the row or column size. Any GPU has only hard-limited number of threads that can execute in parallel. Others are queued, and thus parts of the whole kernel run are executed sequentially. Nvidia added several helper functions to make reduction usage easier, but the functions are available only on devices with compute capability ≥ 3.0 . The following is applied to the PSO code:

- To minimize the PSO execution time (as any algorithm on GPU), any loops whose iterations do not depend on each other must be executed as parallel threads, which means each iteration should be a thread.
- The PSO main loop specifies multiple algorithm runs which do not depend on each other at all to accumulate all results and maybe later take some $\min/\max/\text{avg}$ - which is not time-critical even in the sequential version.

Multiple particles, "moving" at each algorithm step - again no mutual dependency. The only dependency - is the "best" solution at the end of the current iteration between all particles in one run. Therefore `multimin`, `multexts` appear as they process all algorithm runs at the same time, but without dependency all calculations are isolated inside the part of the data arrays associated with each algorithm run. One problem that might greatly slow down the solution is the random numbers generation. Fortunately, PSO does not require pseudo-random numbers to be generated completely sequentially. So, it was enough to specify different seeds for each thread requiring random numbers generation. Surely, the GPU PSO solution is not exactly equal with CPU solution because of the different pseudo-random numbers. If exact numeric matching

between GPU and CPU version would be requirement, the GPU code would be 100-1000 times slower than CPU.

3.2.2. Classes

Classes are collections of data related to a single object type. Classes not only include information regarding the real-world object but also functions to access the data. Classes possess the ability to inherit from other classes. Carefully designed C++ constructors and destructors can go a long way towards easing the pain of manual memory management. Objects can know how to deallocate all their associated resources, including dependent objects (by recursive destruction). This means that clients of a class library do not need to worry about how to free resources allocated on their behalf. With automatic memory management, such as garbage collection, modern techniques can give guarantees about interactive pause times, and so on.

3.2.3. Constructors and Destructors

Classes must always contain two functions: a constructor and a destructor. The syntax for them is simple: the class name denotes a constructor, a ~ before the class name is a destructor. The basic idea is to have the constructor initialize variables and to have the destructor clean up after the class, which includes freeing any memory allocated. If it turns out that we do not need to perform any initialization, then we can allow the compiler to create a "default constructor". When the programmer declares an instance of the class, the constructor will be automatically called. The only time the destructor is called is when the instance of the class is no longer needed, either when the program ends the class reaches the end of scope, or when its memory is deallocated using delete. The key idea is that destructors are always called when the class is no longer usable. Neither constructors nor destructors return arguments.

Objects generally need to initialize variables or assign dynamic memory during their process of creation to become operative and to avoid returning unexpected values during their execution. A class can include a special function called constructor, which is automatically called whenever a new object of this class is created. This constructor function must have the same name as the class and cannot have any return type not even void. Constructors cannot be called explicitly as if they were regular member functions. They are only executed when a new object of that class is created. You can also see how neither the constructor prototype declaration (within the class) nor the latter constructor definition include a return value not even void. Constructor simply sets them to zero (meaning an empty array). Please note that in C++ such (or any other kind of) initialization is required, otherwise data members will be filled with trash. Destructor (`~gpu_array`) is C++ "convenience function". It is automatically called by the language when a variable of that type goes out of scope. `gpu_array` contains 2 data members pointer to data in GPU memory and the number of elements in that array, `exc_gpuarray` is formal C++ definition which is an empty structure. It will be used when our code needs to generate an exception. Generally, you can define as many structures for exceptions as you need. They may contain some data, inherit from standard library prototype etc. etc. We do not need it in the scope of our work. You can later later look at the PSO code to see more functional exception classes.

The destructor fulfills the opposite functionality. It is automatically called when an object is destroyed either because its scope of existence has finished (for example, if it was defined as a local object within a function and the function ends) or because it is an object dynamically assigned and it is released using the operator `delete`. Destructors are special member functions of the class required to free the memory of the object whenever it goes out of scope. Destructors are parameter less functions. Name of the Destructor should be exactly the same as that of the name

of the class. But preceded by ‘~’ (tilde). Destructors do not have any return type not even void. The Destructor of class is automatically called when an object goes out of scope. The use of destructors is especially suitable when an object assigns dynamic memory during its lifetime and now being destroyed we want to release the memory that the object was allocated to.

3.2.4. Exception Handling in C++

One of the advantages of C++ over C is Exception Handling. There are few advantages of exception handling over traditional error handling. C++ provides following specialized keywords for this purpose.

- ‘Try’ represents a block of code that can throw an exception.
- ‘Catch’ represents a block of code that is executed when an exception is thrown.
- ‘Throw’ is used to throw an exception. Also, used to list the exceptions that a function throws, but does not handle itself. C/C++ have nasty feature that any time abort () may be called from any place, and the program exits without any correct cleanup. In particular abort () is called internally when during processing an exception occurred in the destructor, another exception occurs. The program is written in such a way that exceptions are generated each time when any CUDA function fails. So, if the GPU came into a wrong state cascade execution of destructors might lead to generating exceptions inside another exception.

Advantages:

- Separation of Error Handling code from Normal Code: In traditional error handling codes, there are always if else conditions to handle errors. These conditions and the code to handle errors get mixed up with the normal flow. This makes the code less readable

and maintainable. With try catch blocks, the code for error handling becomes separate from the normal flow.

- Functions/methods can handle any exceptions they choose: A function can throw many exceptions, but may choose to handle some of them. The other exceptions, which are thrown but not caught can be handled by the caller. If the caller chooses not to catch them then the exceptions are handled by the caller of the caller. In C++, a function can specify the exceptions that it throws using the throw keyword. The caller of this function must handle the exception in some way (either by specifying it again or catching it).
- Grouping of error types: In C++, both basic types and objects can be thrown as exception. We can create a hierarchy of exception objects, group exceptions in namespaces or classes, categorize them per types.

C++ library has a standard exception class, which is base class for all standard exceptions. All objects thrown by components of the standard library are derived from this class. Therefore, all standard exceptions can be caught by catching this type. In C++, all exceptions are unchecked. The compiler does not check whether an exception is caught or not. For example, in C++ it is not necessary to specify all uncaught exceptions in a function declaration.

3.2.5. CUBLAS

A Linear Algebra library, which duplicates many functions from the well-known BLAS (Basic Linear Algebra Subprograms) libraries, is provided for performing dense vector and matrix algebra. It automatically uses the GPU, and requires that the data be explicitly managed. Data must reside on the GPU before the CUBLAS function is invoked. Most vector or matrix results automatically remain on the GPU, and they must be explicitly moved to the host if needed

there. Some scalar results (e.g. DOT product) can be automatically returned to the host. Typical routine naming:

DAXPY= Double precision A Times X plus Y (X, Y are vectors, A is scalar).

DDOT = Double precision DOT product.

CUBLAS is an implementation of BLAS (Basic Linear Algebra Subprograms) on top of the NVIDIA CUDA driver. It allows access to the computational resources of NVIDIA GPUs. The library is self-contained at the API level, that is, no direct interaction with the CUDA driver is necessary. The basic model by which applications use the CUBLAS library is to create matrix and vector objects in GPU memory space, fill them with data, call a sequence of CUBLAS functions, and, finally, upload the results from GPU memory space back to the host. To accomplish this, CUBLAS provides helper functions for creating and destroying objects in GPU space, and for writing data to and retrieving data from these objects. `_kern_transpose23_t` - in 3-dimensional array, swap 2nd and 3rd dimensions. This was necessary because of the largest dataset in the algorithm (Runs * Particles * Dimensions) should have been processed partially by the CUBLAS library (it does very efficient matrix operations). The problem was that CUBLAS uses column-major alignment in matrices, while PSO code uses C/C++ convention (row-major). `transpose23` and `copy_1d` (CPU functions) are calling appropriately. `_kern_*` (GPU functions).

Tesla GPUs can call kernels from within kernels ("dynamic parallelism"). Dynamic parallelism does not add to performance in most cases, which is the same as with built-in reduction functions. So, all kernels are called from CPU-side at one level. Taking this into account may make PSO code simpler and more understandable. `struct pso_plan` this is a wrapper object for the PSO code. It contains vectors, scalar parameters and matrices with input, intermediate and output data. The chosen target function is calculated at once for all particles in

all "runs" per algorithm step. From the other side, I avoided “deeper” optimization: some target functions might have been parallelized by the number of dimensions. In case where calculations in each dimension are heavy, but at the end only the sum or product is required - this would gain a bit of performance. But this is considerably more complex to implement.

`b_trans()` - true if any of CUBLAS functions must be used (depends on the input parameters of PSO). If so, particles data (multiple matrices) are transposed when CUBLAS is needed and then transposed back. Transposition is rather a cheap operation because of parallelism. `cublasSgemm1`, `cublasSgeam1` calls - they call CUBLAS, but specify that arguments should be transposed before use. So, in this case transposition is built into CUBLAS itself. Our `transpose23 ()` is needed in another place where the array contains multiple matrices. In general, `_test_function` does one of two cases: a) passes `_pos` (particle positions) to the chosen target function, b) applies some transformations to `_x2`, which is a copy of `_pos`, and passes the result to the chosen target function.

4. EXPERIMENTS AND RESULTS

The experiments that were conducted as part of this project proves the theory discussed above. Our experiments were conducted on multiple values of the dimension and particles in which different comparison of values are observed for both CPU and GPU [25]. Both scaling experiments were performed on the CPU and GPU, and we used the Ackley Benchmark function. The mathematical equation is shown in Equation 3:

$$f(\mathbf{x}) = -a \exp \left(-b \sqrt{\frac{1}{d} \sum_{i=1}^d x_i^2} \right) - \exp \left(\frac{1}{d} \sum_{i=1}^d \cos(cx_i) \right) + a + \exp(1) \quad (3)$$

The Ackley function is used for testing the PSO algorithm implementations. In its two-dimensional form, as shown in Figure 11, the outer region is nearly flattened, and at center with a large hole. Some algorithms like hill climbing can get trapped in one of its many local minima. The function is usually evaluated on the hypercube $x_i \in [-32.768, 32.768]$, for all $i = 1, \dots, d$, although it may also be restricted to a smaller domain [25].

Experimental Framework:

GPU: Tesla K40 GPU having 2880 cores

Processor core clock: Base Clock: 745 MHz, Boost Clocks: 810 MHz and 875 MHz

CUDA version: 7.0

OS: Ubuntu 14.04.5 LTS

Input:

fun - target function

R - number of independent test runs (≥ 1)

P - number of particles (≥ 1)

D - number of test function argument dimensions (≥ 1)

L - number of iterations in the algorithm (≥ 2)

xmin, xmax - argument limits, by any dimension

b_norm - non-zero value of b_norm enables normalized positions (internally)

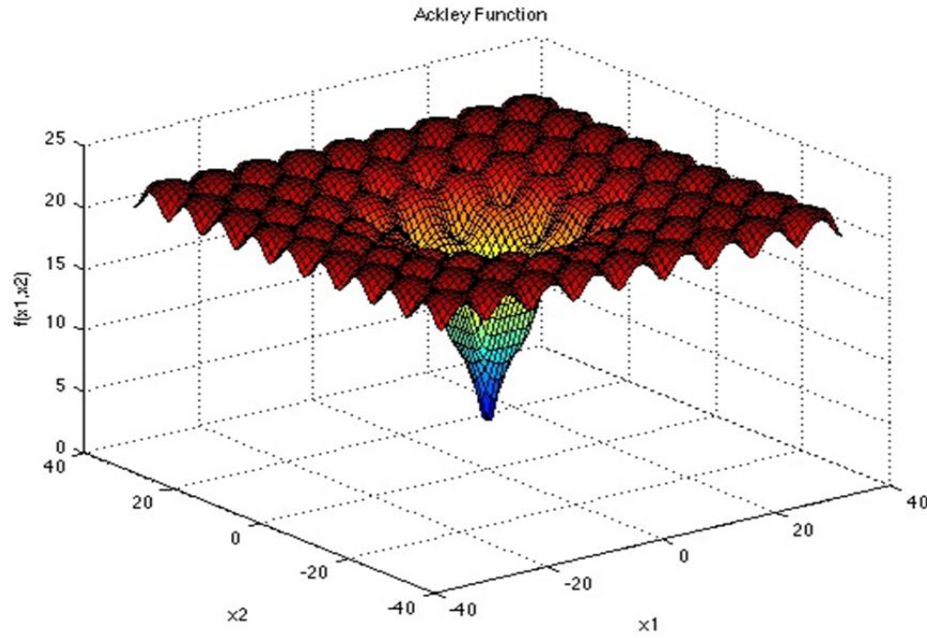


Figure 11: Ackley Function Graphical Representation [26]

The comparison of the two experiments is that the original PSO is run on the CPU of the GPU machine but with normal MATLAB parameters, and the modified code with CUDA is run on the GPU. The experiments vary the number of particles as well as the dimension, respectively, measuring the execution time of the CPU and the GPU version for comparison. Furthermore, large-scale experiments with up to 600 particles and dimensions between 10 and 150 were conducted.

4.1. Scaling Experiment 1

The following parameters are used for the first experiment: Dimension = 10, Number of iterations = 3,000, Vary the number of particles from 50 to 600 in steps of 50. Figure 12 and Figure

13 shows the time taken to run the PSO code with varying number of particles in CPU and GPU respectively.

4.1.1. CPU Time Varying p

Running through the scaling experiments with the Matlab CPU implementation, the following results shown in Table 1 are observed with varying number of particles (i.e. number of particles=50,100,150,200,250,300,350,400,450,500,550,600).

Table 1: CPU time on varying p values

Number of particles	CPU Time (sec)
50	253.934
100	497.145
150	743.775
200	987.970
250	1228.696
300	1482.790
350	1768.568
400	1913.759
450	2112.000
500	2355.264
550	2555.949
600	2878.390

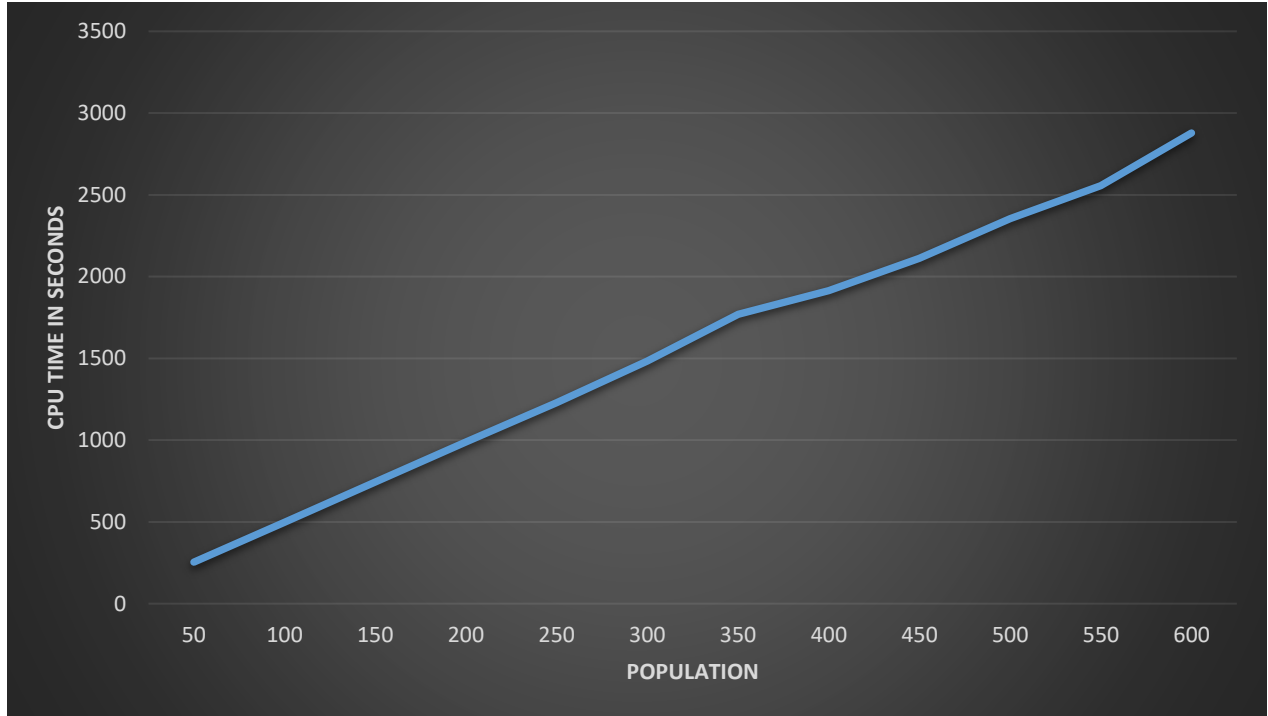


Figure 12: CPU Time Versus Number of Particles with Varying P

Here we observe that as the particle size is increased the time taken by the CPU code has a uniform growth. The figure shows the time taken for the PSO code with varying number of particles when all other parameters are kept constant. In Figure 12, we can see that by increasing the number of particles the time taken to complete the algorithm run is considerably increasing.

Result: The result of this experiment is that by increasing the number of particles the time taken to run the PSO code is uniform and increasing in CPU mode.

4.1.2. GPU Time Varying p

Running through the scaling experiments with GPU CUDA code implementation using Tesla K40 the following results, shown in Table 2, are observed with varying number of particles (i.e. number of particles=50,100,150,200,250,300,350,400,450,500,550,600).

Table 2: GPU time on varying p values

Number of particles	GPU Time (sec)
50	17.761
100	17.677
150	17.780
200	17.875
250	18.000
300	18.080
350	18.190
400	18.491
450	18.621
500	18.691
550	18.812
600	18.914

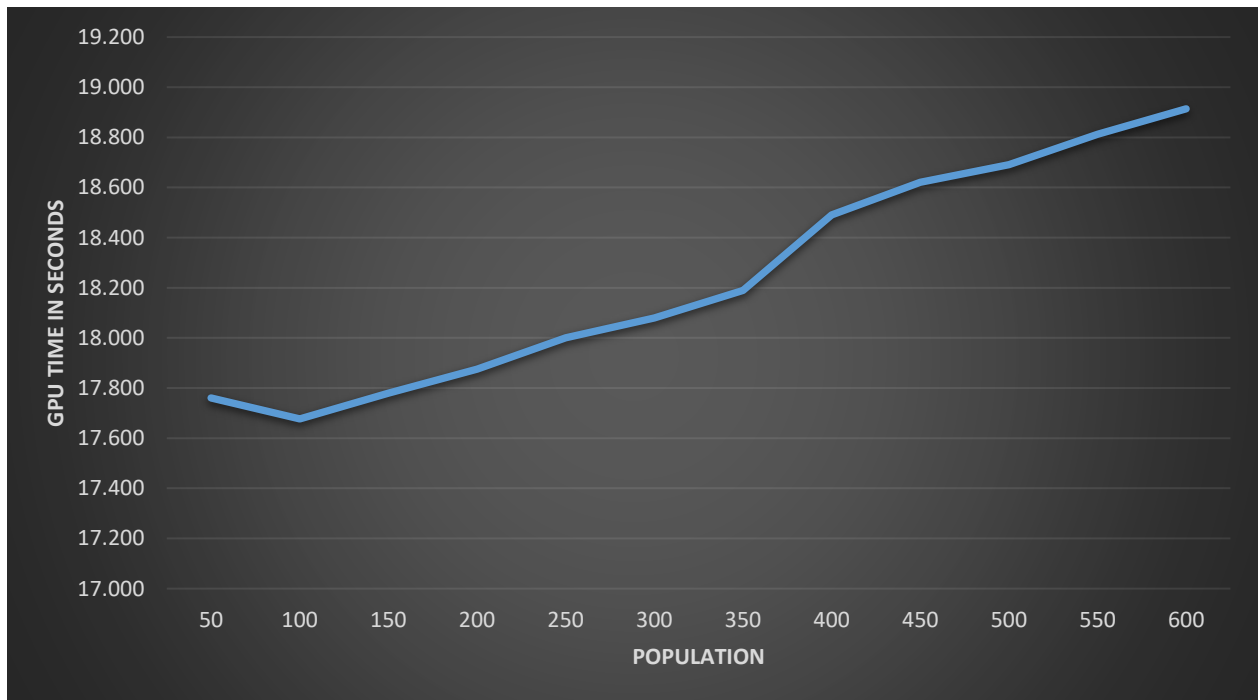


Figure 13: GPU Time Versus Number of Particles on Varying P

Here we observe that as the particle size is increased the time taken by the GPU CUDA code has a uniform growth. Figure 13 shows the time taken for the CUDA PSO code with varying number

of particles when all other parameters are kept constant. In the figure we can see that by increasing the number of particles the time taken to complete the work is considerably increasing.

Result: The result of this experiment is that by increasing the number of particles the time taken to run the PSO code is uniform and increasing for the GPU CUDA code.

4.1.3. CPU and GPU Time Comparison Varying p

Below table 3 shows the speed up value on comparison of the above values of GPU and CPU. The execution time of running the PSO implementations of the CPU (sec) and GPU (sec) will be measured. Furthermore, the speedup will be calculated by: $\text{Speedup} = \text{CPU}(\text{sec}) / \text{GPU}(\text{sec})$.

Table 3: Varying P value CPU vs GPU time with speedup value

Number of particles	CPU (sec)	GPU (sec)	Speedup
50	253.934	17.761	14.56
100	497.145	17.677	28.12
150	743.775	17.780	41.83
200	987.970	17.875	55.27
250	1228.696	18.000	68.26
300	1482.790	18.080	82.01
350	1768.568	18.190	97.23
400	1913.759	18.491	104.11
450	2112.000	18.621	113.42
500	2355.264	18.691	126.01
550	2555.949	18.812	135.87
600	2878.390	18.914	152.18

Table 3 lists the execution time of the CPU and GPU implementations as well as the calculated speedup for varying population sizes. The values show an increasing speedup trend starting from 14.56 up to 152.18. The speedup values are shown in Figure 14 and Figure 15 shows the comparison between the CPU and GPU time performance for different values of p.

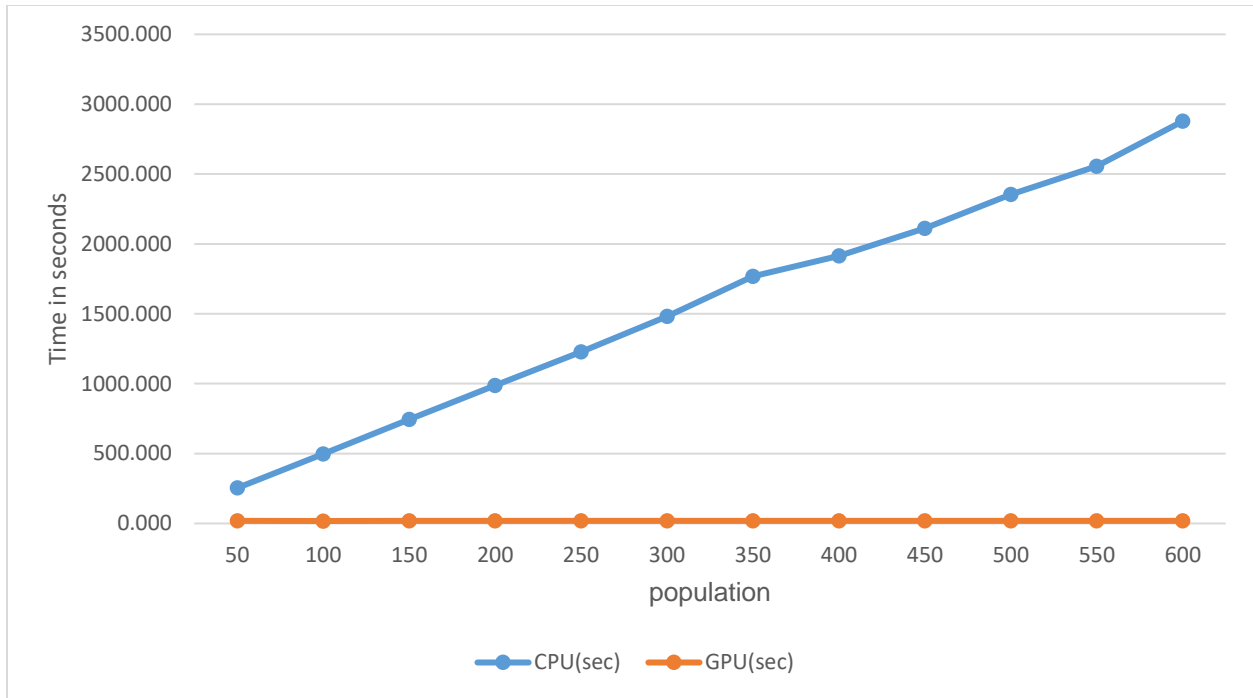


Figure 14: CPU vs GPU Time Performance with Varying P Value

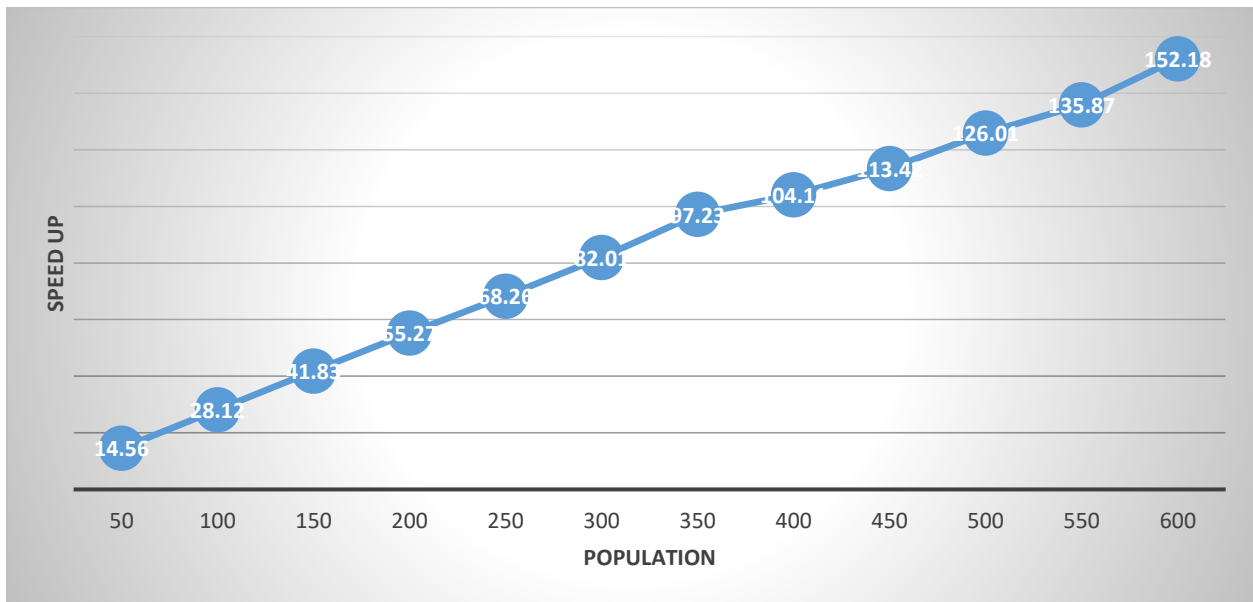


Figure 15: Speedup of GPU Versus CPU with Varying P Value

Result: The values show that the speedup is increasing with increases in the number of particles. The result of this experiment is that by increasing the number of particles the time taken to run the PSO code is observed in the CPU and GPU execution, and the GPU shows better results than the CPU.

4.2. Scaling Experiment 2

The following parameters are used for the second experiment: Number of particles = 50, Number of iterations = 3,000, Vary the dimension from 10 to 150 in steps of 10. Figure 16 and Figure 17 shows the time taken to run the PSO code with varying number of dimension in CPU and GPU, respectively.

4.2.1. CPU Time Varying d

Running the scaling experiments with the MATLAB CPU implementation on the GPU machine, the following results, shown in Table 4, are observed with varying number of dimensions (i.e. number of dimensions=10,20,30,40,50,60,70,80,90,100,110,120,130,140,150)

Table 4: CPU time on varying d values

Number of Dimensions	CPU (sec)
10	268.400
20	267.770
30	283.451
40	293.180
50	307.540
60	312.400
70	333.158
80	347.940
90	355.480
100	460.912
110	480.419
120	498.295
130	508.138
140	524.526
150	523.670

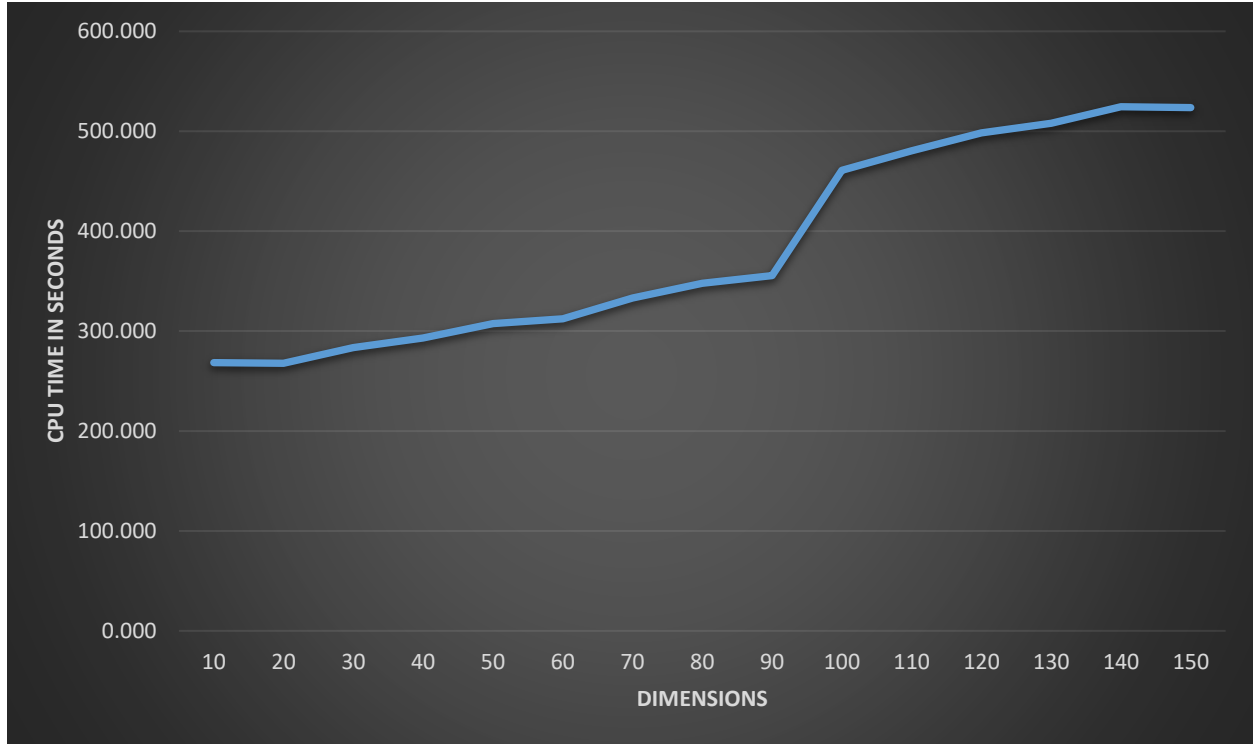


Figure 16: CPU Time Versus Number of Dimensions on Varying D

Here we observe that as the dimension size is increased the time taken by the CPU code has a uniform growth. Figure 16 shows the time taken for the CPU with varying number of dimensions when all other parameters are kept constant. In the above figure, as we can see by increasing the number of dimensions the time taken to complete the optimization run is considerably increasing.

Result: The result of this experiment is that by increasing the number of dimensions the time taken to run the PSO code is uniform and increasing in the CPU mode.

4.2.2. GPU Time Varying d

Running the scaling experiments with the GPU CUDA code using Tesla K40, the following results shown in Table 5 are observed with varying number of dimensions (i.e. number of dimensions= (10,20,30,40,50,60,70,80,90,100,110,120,130,140,150)).

Table 5: GPU time on varying d values

D Value	GPU (sec)
10	17.610
20	17.694
30	17.772
40	17.868
50	17.982
60	18.073
70	18.194
80	18.268
90	18.379
100	18.479
110	18.526
120	18.609
130	18.727
140	18.873
150	18.983

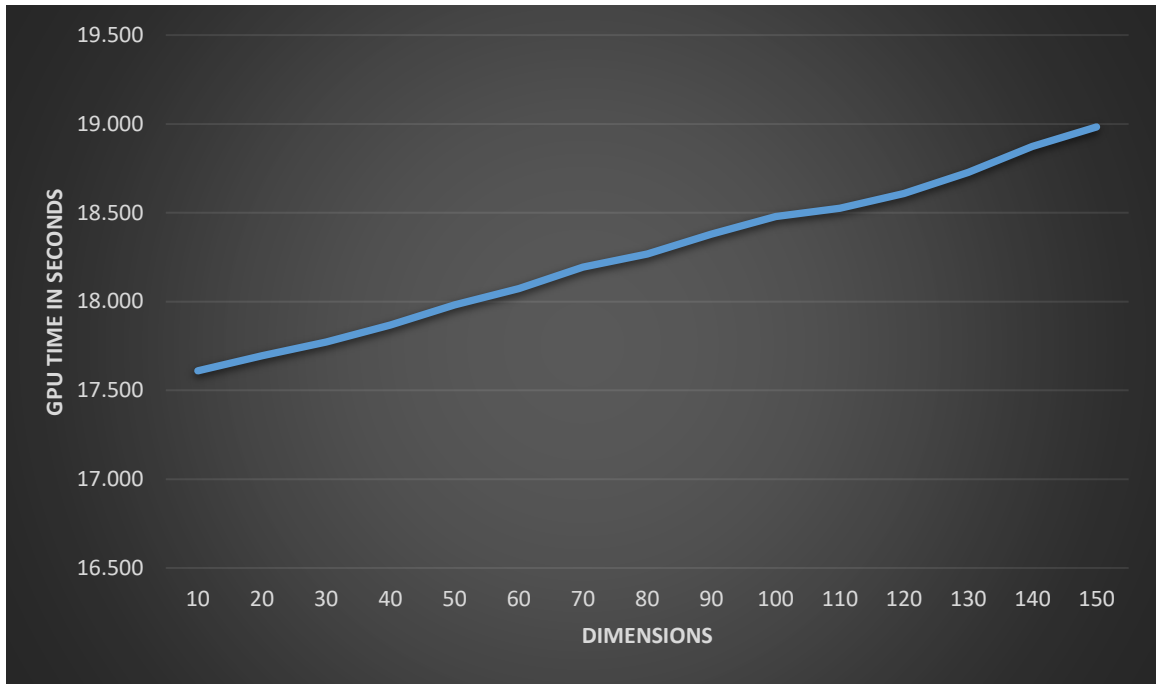


Figure 17: GPU Time Versus Number of Dimensions on Varying D Values

Here, we observe that as the number of dimensions are increased the time taken by the GPU CUDA code has a uniform growth. Figure 17 shows the time taken for the CUDA PSO code with

varying number of dimensions with all other parameters kept constant. As we can see by increasing the number of dimensions the time taken to complete the work is considerably increasing.

Result: The result of this experiment is that by increasing the number of dimensions the time taken to run the PSO code is uniform and increasing for the GPU CUDA code.

4.2.3. CPU and GPU Time Comparison Varying d

Table 6 shows the speedup value on comparison of the above values of GPU and CPU. The execution time of running the PSO implementations of the CPU (sec) and GPU (sec) will be measured. Furthermore, the speedup will be calculated by: $\text{Speedup} = \text{CPU}(\text{sec}) / \text{GPU}(\text{sec})$.

Table 6: Varying d value CPU vs GPU time with speedup value

D value	CPU (sec)	GPU (sec)	Speedup
10	268.400	17.610	15.13
20	267.770	17.694	15.25
30	283.451	17.772	15.95
40	293.180	17.868	16.41
50	307.540	17.982	17.1
60	312.400	18.073	17.29
70	333.158	18.194	18.31
80	347.940	18.268	19.05
90	355.480	18.379	19.34
100	460.912	18.479	25.01
110	480.419	18.526	25.93
120	498.295	18.609	26.78
130	508.138	18.727	27.13
140	524.526	18.873	27.79
150	523.670	18.983	28.06

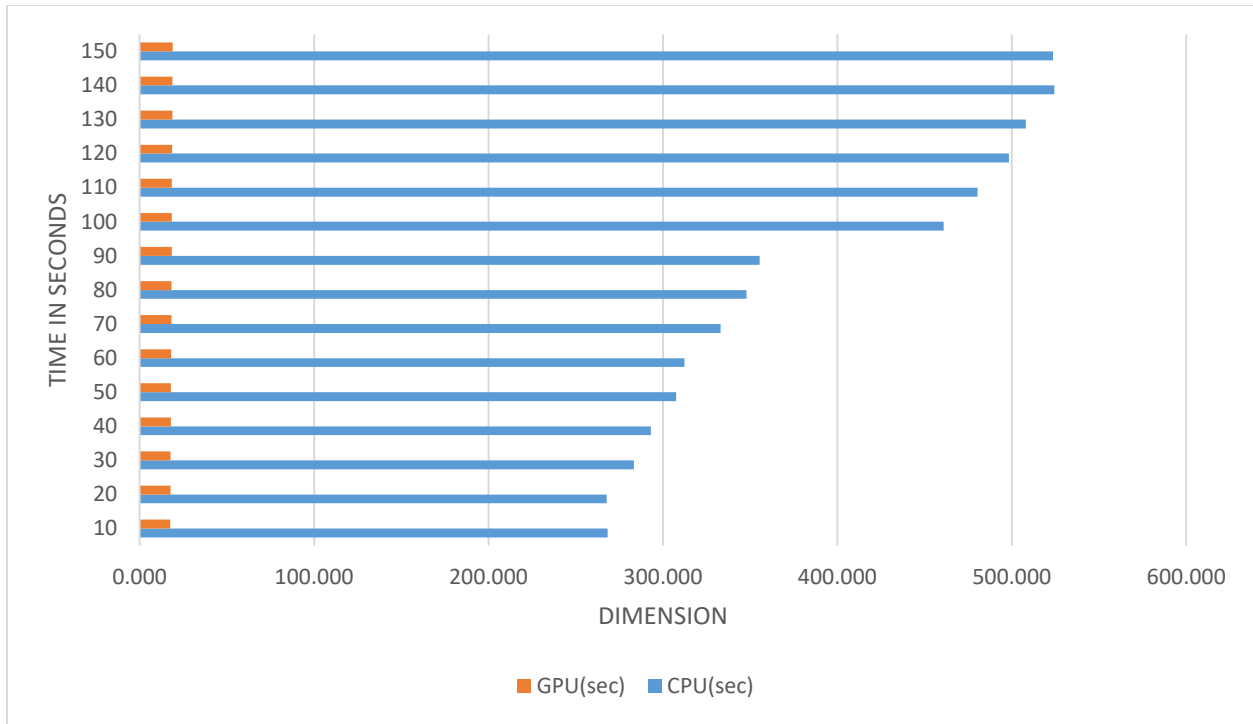


Figure 18: CPU vs GPU Time Performance with Varying D Value

Table 6 lists the execution time of the CPU and GPU implementations as well as the calculated speedup for varying population sizes. Figure 18 shows the comparison of the CPU and GPU times with varying d value. The values show an increasing speedup trend from 15.13 to 28.06. The values are shown in Figure 19.

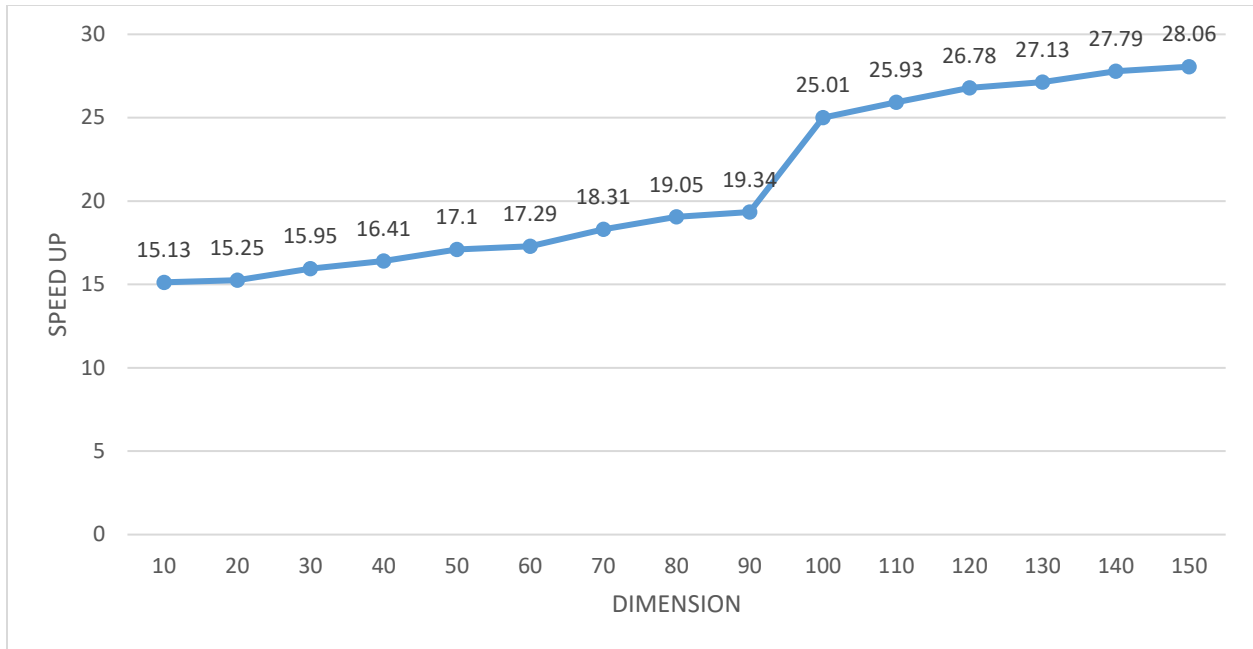


Figure 19: Speed Up of GPU Versus CPU on Varying D Value

Result: The values show that the speed up is constant throughout the change in dimensions. The result of this experiment is that by increasing the number of dimensions the time taken to run the PSO code is observed for both the CPU and GPU implementation, and the GPU shows better results than CPU.

5. CONCLUSION AND FUTURE WORK

With these experiments, we have proved that with the GPU device like Tesla K40, a search method such as Particle Swarm Optimization can be solved within a significantly less amount of time when compared to when executed on a CPU. Experiments were conducted to demonstrate the scalability gains obtained on a Tesla K40. We measured the execution time of the CPU and the GPU version to compare their results on varying numbers of particles as well as the dimension. Also, a few other experiments looked at the scalability of the GPU implementation, again scaling the number of particles as well as the dimension. Some results for the experiments are as the GPU implementation has a shorter runtime than the CPU implementation for a PSO run with many dimension as well as a large population size.

As for future work, we would like to implement the PSO algorithm further by experimenting with different kernel function. Furthermore, this work only evaluated the CPU and GPU implementation of the PSO algorithm on a single benchmark function and thus a more thorough analysis with other costlier benchmark functions might be interesting to conduct. With this I conclude that using GPU for complex problems on multicore processor will improve the time taken to complete it with promising results.

6. REFERENCES

1. Stout, Q. F. (1994). Behind the scenes of HPCC, in Suggesting Computer Science Agendas for High Performance Computing. In U. vishkin (Ed.), Computing Machinery.
2. Boukhanovsky, A. V., & Ivanov, S. V. (2003). Stochastic simulation of inhomogeneous metocean fields. In Melbourne, Australia.
3. Harris, M. J. General Purpose Computation on GPUs. Retrieved from <http://www.gpgpu.org>.
4. Artificial Intelligence Computing Leadership from NVIDIA. (2017). Retrieved from <http://www.nvidia.com/content/global/global.php>.
5. Zhou., Y., & Tan., Y. (2011). GPU-based Parallel Multi-Objective Particle Swarm Optimization. In (Vol. 7): International Journal of Artificial Intelligence.
6. al, J. W. e. (2015). GPU-based Adaptive Mutation PSO Algorithm. In: Journal of Computational Information Systems.
7. Mussi, L., Nashed, Y. S. G., & Cagnoni, S. (2011). GPU-based Asynchronous Particle Swarm Optimization. In. Dublin, Ireland: Proceedings of Genetic and Evolutionary Computation Conference (GECCO).
8. Dali, N., & Bouamama, S. (2015). GPU-PSO: Parallel Particle Swarm Optimization Approaches on Graphical Processing Unit for Constraint Reasoning. In (Vol. 60): Case of Max-CSPs, Procedia Computer Science.
9. Roberge., V., & M.Tarbouchi. (2012). Parallel particle swarm optimization on graphical processing unit for pose estimation. In (pp. 170-179): WSEAS Transactions on Computers.

10. Nagano, K., Collins, T., Chen, C.-A., & Nakano., A. (2015). GPU-based inverse rendering with multi-objective particle swarm optimization. In. New York, NY, USA: SIGGRAPH Asia 2015 Visualization in High Performance Computing.
11. al, J. W. e. (2015). Parallel hybrid PSO with CUDA for 1D heat conduction equation, Journal of Computers & Fluids. In (Vol. 110): Elsevier.
12. Veronese, L. d. P., & Krohling., R. A. (2009). Swarm's Flight: Accelerating the Particles using C-CUDA. In: IEEE Congress on Evolutionary Computation (CEC).
13. Wenna, L., & Zhenyu, Z. (2011). A CUDA-based Multi-Channel Particle Swarm Algorithm. In: Proceedings of Int. Conf. on Control, Automation and Systems Engineering(CASE).
14. You, Z., & Ying, T. (2009). GPU-based parallel particle swarm optimization. In: IEEE Congress on Evolutionary Computation (CEC).
15. Mussi, L., & Cagnoni., S. (2009). Particle Swarm Optimization within the CUDA Architecture. In: 11th Annual Conference on Genetic and Evolutionary Computation (GECCO).
16. Brodtkorb.A, Dyken.C, Hagen.T., Hjelmervik.J., & Storaasli.O. (2010). State-of-the-art in heterogeneous computing. In: Sci. Program.
17. CUDA C programming guide version 6.5. (2014). In: NVIDIA Corporation.
18. GPU Architecture. Retrieved from <https://people.duke.edu/~ccc14/sta-663/CUDAPython.html>.
19. CUDA C Programming Guide. Retrieved from <https://docs.nvidia.com/cuda/cuda-c-programming-guide/>.

20. Wang, L., Li, S., & Zhang, G. A GPU-Based Parallel Procedure for Nonlinear Analysis of Complex Structures Using a Coupled FEM/DEM Approach. In Volume 2013 (2013), Article ID 618980, Mathematical Problems in Engineering.
21. NVIDIA's Next Generation CUDA™ Compute Architecture: Kepler™ GK110. Retrieved from <https://www.nvidia.com/>.
22. NVIDIA CUDA Programming version 6.0. (2014). In. Retrieved from <https://www.nvidia.com/>.
23. Sanders, J., & Kandrot, E. Cuda by Example. In: Nvidia. Retrieved from <https://docs.nvidia.com/cuda/cuda-c-programming-guide/>.
24. Farber, R. Cuda Application Design and Development. In: Nvidia. Retrieved from <https://www.nvidia.com/>.
25. Adorio, E. P., & Diliman, U. P. M. (2005). Multivariate Test Functions Library in C for Unconstrained Global Optimization. In. Retrieved from <http://www.geocities.ws/eadorio/mvf.pdf>
26. Molga, M., & Smutnicki, C. (2005). Test functions for optimization needs In Using an Evolutionary Heuristics for Solving the Outdoor Advertising Optimization Problem. Journal of Computer Sciences and Applications. In. Retrieved from <http://www.zsd.ict.pwr.wroc.pl/files/docs/functions.pdf>.